HEPACCELERATE: FAST ANALYSIS OF COLUMNAR COLLIDER DATA

A PREPRINT

J. Pata, M. Spiropulu
California Institute of Technology

June 17, 2019

ABSTRACT

At HEP experiments, processing terabytes of structured numerical event data to a few statistical summaries is a common task. This step involves selecting events and objects within the event, reconstructing high-level variables, evaluating multivariate classifiers with up to hundreds of variations and creating thousands of low-dimensional histograms. Currently, this is done using multi-step workflows and batch jobs. Based on the CMS search for $H(\mu\mu)$, we demonstrate that it is possible to carry out significant parts of a real collider analysis at a rate of up to a million events per second on a single multicore server with optional GPU acceleration. This is achieved by representing HEP event data as memory-mappable sparse arrays, and by expressing common analysis operations as kernels that can be parallelized across the data using multithreading. We find that only a small number of relatively simple kernels are needed to implement significant parts of this Higgs analysis. Therefore, analysis of real collider datasets of billions events could be done within minutes to a few hours using simple multithreaded codes, reducing the need for managing distributed workflows in the exploratory phase. This approach could speed up the cycle for delivering physics results at HEP experiments. We release the `hepaccelerate` prototype library as a demonstrator of such accelerated computational kernels. We look forward to discussion, further development and use of efficient and easy-to-use software for terabyte-scale high-level data analysis in the physical sciences.

1 Introduction

At the CMS experiment of the Large Hadron Collider, the final stage of data processing is typically carried out over several terabytes of numerical data residing on a shared cluster of servers used for batch processing. The data consist of columns of features for the recorded particles such as electrons, muons, jets and photons for each event in billions of rows, one row per event. In addition to the columns of purely kinematic information of particle momentum, each particle carries a number of features that describe the reconstruction details and other high-level properties of the reconstructed particles. For example, for muons, we might record the number of tracker layers where an associated hit was found, whether or not it reached the outer muon chambers and in MC simulation the index of the associated generator-level particle, thereby cross-linking two collections. Typical compressed event sizes for such reduced data formats at CMS are a few kilobytes per event. In practice, mixed precision floating points are used to store data only up to experimental precision, such that the number of features per event is on the order of a few hundred.

A typical physics analysis at the LHC such as the precision measurement of a particle property or the search for a new phenomenon involves billions of real and simulated events. The final result typically requires tens to hundreds of iterations over this dataset while the analysis is ongoing over the period of months to a year. For each iteration of the analysis, hundreds of batch jobs of custom reduction software is run over these data. The maintenance and operation of this data pipeline takes up valuable researcher time and can result in delays between iterations of the analysis, slowing down innovation in experimental methods.

We demonstrate that by efficiently accessing and caching only the needed columns from the data, sizeable batches of data can be loaded from disk to memory at the speed of several MHz (millions of events per second). By processing the data using efficient vectorizable kernels on the arrays, applying multithreading and GPU acceleration to the final

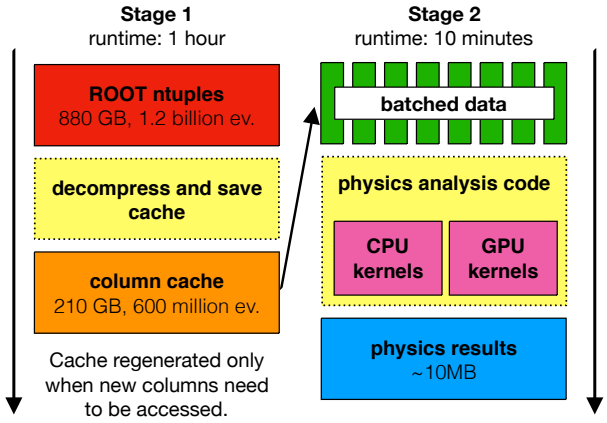

Figure 1: The flowchart of the accelerated analysis workflow for the benchmark CMS $H(\mu\mu)$ analysis. The necessary ROOT data are accessed from a networked filesystem with a total size of about 880 GB. In the caching step, the relevant feature columns are decompressed and extracted and optional preselection is applied. The caching step takes approximately 2 hours for a 24-core job processing 1.2 billion events and is largely limited by IO. The cache consists of about 200 GB of data, 600M events of memory-mappable contiguous arrays and is portable between different platforms. This analysis-specific cache can be processed with the $H(\mu\mu)$ selection in 5 minutes on a 24-core server and in about 20 minutes on a commodity laptop.

stage of data analysis becomes natural, such that data can also be processed at MHz-level speeds already on a single server. This means that a complete physics analysis of billions of events can be run as a single integrated Python code on a single machine with a rate of about a billion events per hour. Although we use a specific and relatively simple CMS analysis as an example, the method based on array computing with accelerated kernels is generic and can be used for other collider analyses. The purpose of this paper is to detail the issue of processing terabyte-scale data in HEP efficiently. We release the `hepaccelerate` library for further discussion [2].

In the following parts, we explore this approach based on columnar data analysis in more detail. In section 2, we describe the structure of the data and discuss how data sparsity is handled efficiently. We introduce physics-specific computational kernels in section 3 and describe the measured performance under a variety of conditions in section 4. Finally, we conclude with a summary and outlook in section 5.

2 Data structure

We can represent HEP collider data in the form of two-dimensional matrices, where N rows correspond to events and columns correspond to features in the event such as the momentum components of all measured particles. However, due to the nature of the underlying physics processes that produce a varying number of particles per event, the number of features varies from one collider event to another, such that a fixed-size two dimensional representation is not memory-efficient. Therefore, the standard HEP software framework based on ROOT includes mechanisms for representing dynamically-sized arrays as well as complete C++ classes with arbitrary structure as the feature columns and a mechanism for serializing and deserializing these dynamic arrays [1].

Based on the approach first introduced in the `uproot` [3] and `awkward-array` [4] python libraries, many existing HEP data files with a varying number of particles per event can be represented and efficiently loaded as sparse arrays with an underlying one-dimensional array for a single feature. Event boundaries are encoded in an offset array that records the particle count per event. Therefore, the full particle structure of events can be represented by a contiguous offset array of size N and a contiguous offset array of size M for each particle feature. This can easily be extended to event formats where the event contains particle collections of different types, e.g. jets, electrons and muons. By using the data and offset arrays as the basis for computation, efficient computational kernels can be implemented and evaluated on the data. We illustrate the jagged data structure on figure 2.

In practice, analysis-level HEP event data are stored in compressed ROOT files storing raw features in a so-called "flat analysis ntuple" form. Out of hundreds of stored features, a typical analysis might use approximately 50 to 100, discarding the rest and only using them rarely for certain auxiliary calibration purposes. When the same features are

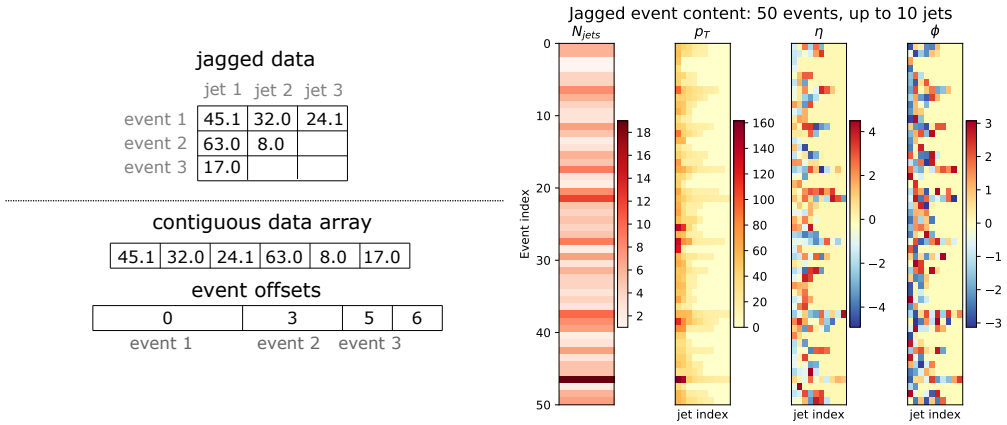

Figure 2: A visual representation of the jagged data structure of the jet p_T, η and ϕ content in 50 simulated events. On the diagram on the left, we illustrate how three events with a varying number of jets are recorded as contiguous arrays. On the figure on the right, we show the number of jets per event, one event per row, which is derived from the offset array. In the three rightmost columns, we show the jet content in events, visualizing the p_T, η and ϕ of the first 10 jets for each event.

accessed multiple times, the cost of decompressing the data can be significant. Therefore, in order to maximize the computational efficiency of the analysis, we have implemented a simple cache based on memory mapped files for the feature and offset data. The efficiency of the uncompressed cache depends on the ratio of features used for analysis. We find that for the representative CMS analysis, the average uncompressed cache size is approximately 0.5 kB/event after choosing only the necessary columns, down from approximately 2 kB/event in compressed form. The choice of compression algorithms can and should be addressed further in optimizing the file formats for cold storage and analysis [6].

3 Computational kernels

In the context of this report, a kernel is a function that is mapped across an array to transform the underlying data. A simple kernel could compute the square root of all the values in the array. More complicated kernels such as convolutions might involve relations between neighbouring array elements or involve multiple inputs and outputs of varying sizes. When the individual kernel calls across the data are independent of each other, these kernels can be evaluated in parallel over the data using SIMD processors. The use of efficient computational kernels with an easy-to-use API has proven to be successful for machine learning frameworks such as `tensorflow`. Therefore, we propose to implement HEP analyses similarly using common kernels that are steered from a single high-level code.

In the following, we will demonstrate that by using the data and offset arrays as described in section 2, common HEP operations such as computing the total scalar sum of momenta of all selected particles in an event can be formulated as kernels and dispatched to vector processing units (CPUs, GPUs), efficiently processing the event data in parallel. The central result of this paper is that only a small number of simple kernels, easily implemented in e.g. Python or C, are needed to implement a realistic HEP analysis.

As mentioned above, a prototypical HEP-specific kernel would be to find the scalar sum H_T of all particles passing some quality criteria in an event. We show the Python implementation for this on listing 1. This kernel takes as input the M-element data array of all particle transverse momenta `pt_data` and an $N+1$-element array of the event offsets. In addition, as we wish to include only selected particles in selected events in this analysis, we use an N-element boolean mask for events and M-element boolean mask for the particles that have passed selection. These masks can be propagated to further functions, making it possible to efficiently chain computations without resorting to expensive data copying. Finally, the output is stored in a preallocated array of size N that is initialized to zeroes.

```
1  def sum_ht(
2    pt_data, offsets,
3    mask_rows, mask_content,
4    out):
5
6    N = len(offsets) - 1
7    M = len(data)
8
9    #loop over events in parallel
10   for iev in prange(N):
11     if not mask_rows[iev]:
12       continue
13
14     #indices of the particles in this event
15     i0 = offsets[iev]
16     i1 = offsets[iev + 1]
17
18     #loop over particles in this event
19     for ielem in range(i0, i1):
20       if mask_content[ielem]:
21         out[iev] += pt_data[ielem]
```

Listing 1: Python code for the kernel computing the scalar sum of selected particle momenta H_T. The inputs are `pt_data`, an M-element array of p_T data for all the particles, the N-element `offsets` array with the indices between the events in the particle collections, as well masks for events and particles that should be considered. On line 10, the kernel is executed in parallel over the events using the Numba `prange` iterator, which creates multithreaded code across the loop iterations. On line 19, the particles in the event are iterated over sequentially.

This kernel generically reduces a data array using a pairwise operator $(+)$ within the offsets and can be reused in other cases, such as counting the number of particles per event passing a certain selection. Other kernels that turn out to be useful are related to finding the minimum or maximum value within the offsets or retrieving or setting the m-th value of an array within the event offsets.

The generic kernels we have implemented for the $H(\mu\mu)$ analysis are the following:

- `get_in_offsets`: given jagged data with offsets, retrieves the n-th elements for all rows. This can be used to create a contiguous array of e.g. the leading jet p_T for further numerical analysis.
- `set_in_offsets`: as above, but sets the n-th element to a value. This can be used to selectively mask objects in a collection, e.g. to select the first two jets ordered by p_T.
- `sum_in_offsets`: given jagged data with offsets, calculates the sum of the values within the rows. As we have described above, this can be used to compute a total within events, either to count objects passing selection criteria by summing masks or to compute observables such as H_T.
- `max_in_offsets`: as above, but calculates the maximum of the values within rows.
- `min_in_offsets`: as above, but calculates the minimum.
- `fill_histogram`: given a data array, a weight array, histogram bin edges and contents, fills the weighted data to the histogram. This is used to create 1-dimensional histograms that are common in HEP. Extension to multidimensional histograms is straightforward.
- `get_bin_contents`: given a data array and a lookup histogram, retrieves the bin contents corresponding to each data array element. This is used for implementing histogram-based reweighting.

This demonstrates that a small number of dedicated kernels can be used to offload a significant part of the analysis. There are also a number of less generic analysis operations which do not easily decompose into other fundamental array operations, but are still useful for HEP analysis. A particular example would be to find the first two muons of opposite charge in the event, or to perform $\Delta R(\eta, \phi)$-matching between two collections of particles. In the standard approach, the physicist might simply write down procedural code in the form of a nested loop over the particles in the event which is terminated when the matching criterion is satisfied. These functions can similarly be expressed in the form of a dedicated kernels that do a single pass over the data and that are easy to write down and debug procedurally for the physicist before being dispatched to SIMD processors. We choose to implement these specifically rather than

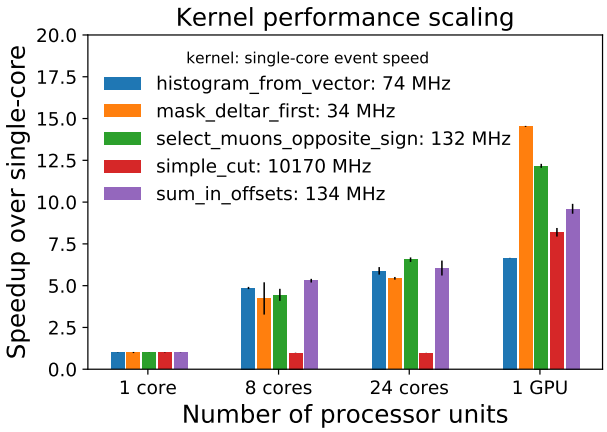

Figure 3: Computational kernels benchmarked on the reference server. We compare the performance of the kernels on approximately 150k preloaded events, showing the scaling with respect to the single-core baseline. The absolute performance is on the scale of tens to hundreds of MHz and is displayed in the legend. In particular, we find that the kernel for computing ΔR masking runs at a speed of 34 MHz on a single-core of the CPU and is sped up by about a factor x5 (x15) by multithreading using the CPU (GPU).

rely on more general array operations that perform cross-collection joins for speed and simplicity of implementation. These specialized kernels are as follows:

- `mask_deltar_first`: given two collections of objects, masks all objects in the first collections that are closer than a predefined value in $\Delta R^2 = \Delta\eta^2 + \Delta\phi^2$ to an object in the second collection
- `select_opposite_sign_muons`: given a collection of objects with charge (i.e. muons), masks all but the first two objects ordered by p_T which are of opposite charge

With a complete set of about 10 kernels, implemented in Python and just-in-time compiled to either multithreaded CPU or GPU/CUDA code using the Numba package [5], a standard HEP analysis such as the search for $H(\mu\mu)$ can be carried out from a simple controller script. We have chosen Python and Numba to implement the kernels in the spirit of quickly prototyping this idea, but this approach is not restricted to a particular programming language. The total number of lines of code for both the CPU and GPU implementations of all kernels is approximately 250, reflecting the simplicity of the implementations. Using a 24 core Intel Xeon E5-2687W v4 @ 3.00GHz with Intel Optane SSDs, networked storage using CephFS and an nVidia Geforce Titan X, we have benchmarked the performance of the kernels on preloaded data. The results are shown on figure 3. For complex kernels such as ΔR masking, with 24 cores we observe a speedup of approximately 5 times over single-core performance. On the GPU, we find a total speedup of approximately 15x over single-core performance.

4 Analysis benchmark

Besides testing the individual kernels in a controlled setting, we benchmark a complete physics analysis based on multithreaded kernels using a CMS analysis for $H(\mu\mu)$. This analysis requires trigger selection, jet and lepton selection, matching of leptons to trigger objects as well as generator-level objects and of jets to leptons. The dimuon candidate is constructed from muons that pass quality criteria. Events are categorized based on the number of additional leptons and jets and the distribution of several control variables as well as the variable of interest, the dimuon invariant mass, is saved to histograms for further analysis. We also evaluate a high-level pretrained DNN discriminator using `tensorflow` based on about 25 input variables. Pileup reweighting is applied on the fly to correct the MC distribution of the number of reconstructed vertices to the one observed in data, with variated histograms being used to account for systematic uncertainties. We use existing CMS code for lepton scale factor and Rochester corrections, which is integrated using Python CFFI and multi-threaded using OpenMP. In general, the most computationally complex steps in high-level analysis are the evaluation of machine learning models and filling of hundreds to thousands of histograms with systematic variations. Both of these are highly efficient on SIMD processors such as GPUs. In order to emulate a more complex analysis, once the data are loaded to memory in batches, we study the scaling behaviour with respect to analysis complexity by rerunning the analysis multiple times on the same data, which reflects the possibility of easily doing parameter scans in this approach.

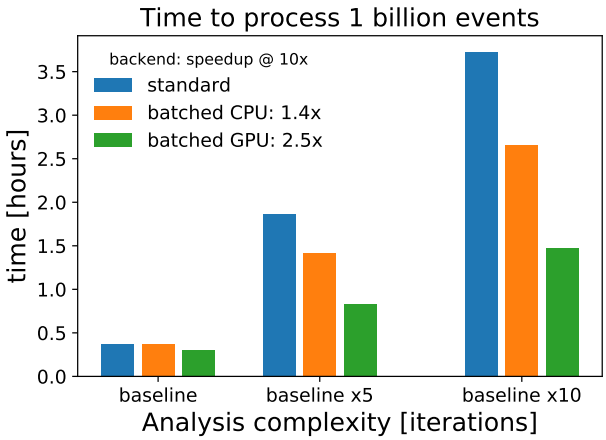

Figure 4: Extrapolated processing time for the $H(\mu\mu)$ analysis on 1 billion events and scaling with respect to analysis complexity. More complex analyses are emulated by iterating same physics code on data already loaded to memory either 5x or 10x. We compare the expected time of a naive reanalysis on the CPU backend (blue) with respect to reanalyzing a loaded batch on the CPU backend (orange) and on the GPU backend (green). A naive reanalysis scales linearly with complexity, whereas analyzing data already in memory ensures the code is compute-bound rather than IO-bound. The GPU backend provides the best scaling with respect to analysis complexity.

We use approximately 90 million MC events in 80 GB of compressed CMS NanoAOD files to benchmark the analysis, reflecting about 10% of the total needed for a single year of data taking. First, we decompress and cache data at a speed of approximately 100-1000 kHz on the benchmark machine, creating an uncompressed cache of approximately 40 GB which is stored on a local SSD. This step is crucial for increased physics throughput, but needs to be repeated only when additional branches need to be included in the analysis. We find that it is advantageous to store the input data on a fast parallel storage with a capacity of several terabytes that supports random reads at speeds exceeding 100 MB/s, but caching speeds from networked bulk storage such as HDFS-FUSE are still acceptable at around 50-100 kHz. The cache creation speed is largely IO limited and can vary significantly depending on the hardware. Further optimization of the decompression and loading step of columnar ROOT data would be beneficial.

After the branch cache is created, the physics analysis code can be run efficiently. We observe baseline event processing rates of approximately 0.8 MHz on the multicore CPU backend and approximately 0.9 MHz using the GPU backend for a single analysis iteration. It is important to note that the absolute event processing speed depends on the analysis complexity. In HEP data analysis it is common to apply some preselection or "skimming" such as trigger bit requirements to the data that are stored or cached for regular reanalysis. By doing this, we avoid loading data that we know will never be used and the effective data processing rate will thus be increased by the preselection factor, which in the case of $H(\mu\mu)$ is approximately three. We choose to report the speed with respect to raw event numbers from CMS simulation without preselection, which would artificially inflate the reported speed. For the baseline analysis, the CPU and GPU backends perform about equally. The GPU backend is advantageous in the case of more complex analyses: when the analysis complexity is scaled up by a factor of 10, we find that the GPU-based backend completes the analysis approximately 2x faster than a pure CPU implementation, and provides a 2.5-fold speedup with respect to a naive implementation. These results are summarized on figure 4. In the implementation of the data flow pipeline, we use a multithreaded approach where a worker thread preloads cached data to system memory and feeds it into a thread-safe queue, which is processed by either the CPU or GPU backend on a separate thread. The batch size of the preloaded data determines the memory usage of the code, which is currently around 10-15 GB total (< 0.7 GB/core). The complete analysis is encapsulated in a single multithreaded python script which can be steered directly from the command line and does not require running additional server software or services.

We have not performed a symmetrical benchmark with respect to current commonly used non-array-based codes for this analysis, as these cannot be easily run from a single integrated workflow and thus cannot be reliably benchmarked end-to-end. However, neglecting the time spent in batch job preparation and management, event processing speeds for ROOT-based event loop analyses are commonly in the range of 50 - 100 kHz, which also factors in data fetching and decompression. We therefore stress that the benchmark shown in this result is not aimed to be symmetrical against standard analysis codes, but rather reflects the possibilities afforded by using array-based techniques on local data for analysis, which allows for fast experimentation and iteration on sizeable datasets before resorting to extensive

Platform	Time to create caches	Time to run physics code
24-core server + GPU	50 minutes	10 minutes
4-core laptop	N/A	30 minutes

Table 1: Time to run the analysis on 1.2 billion raw events on a server [a] and on a laptop [b]. We list the time to decompress and cache approximately 880 GB of ROOT data, which is done only on the server, and the time to run the physics analysis code on approximately 210 GB of uncompressed and preselected caches.

[a] 24-core Intel Xeon E5-2687W v4 3.00GHz, Intel P4608 SSD, nVidia GTX Titan X
[b] 4-core 2.3 GHz Intel i5 laptop, 8GB RAM, Thunderbolt 3 SSD

distributed computing infrastructure. We hope this encourages further development and optimization of HEP data analysis codes along the performance and scaling axes.

In addition to benchmarking the scaling performance on a small dataset of 90 million simulated events as above, we have found that an end-to-end physics processing of roughly a billion events of data and MC is possible within minutes. The input data consist of approximately 1.2 billion events for the 2017 data taking period in approximately 880 GB of ROOT files. After the initial column caching and decompression step, which takes about an hour for a single 24-core job on the benchmark machine, a cache of approximately 210 GB is produced, preselecting 600M events based on the isolated muon trigger bit and saving only the approximately 50 analysis-specific columns. This cache is stored on an SSD for fast access and is portable between machines. The benchmarks are summarized in table 1.

Although some computationally demanding analysis features such as jet energy scale variations are not yet implemented, the scaling behaviour suggests that even a significantly more complex analysis could be completed within a few hours using only a single machine. Both horizontal and vertical scaling are possible. The work can be distributed across multiple machines using data parallelism within environments such as Apache Spark or using the more traditional batch software. Further engineering work on multithreading and kernel optimization, GPU streaming and caching is expected to increase the achievable analysis speeds on a single machine.

5 Summary and outlook

The central message of this report is that it is possible to do significant physics analysis processing from a simple code on a single machine using multithreading within minutes to hours. Using memory-mappable caching, array processing approaches and a small number of specialized kernels for jagged arrays implemented in Python using Numba, it is possible to do real physics analysis tasks with event rates reaching up to a million events per second. It is also possible to offload parts of these array computations to accelerators such as GPUs, which are highly efficient at SIMD processing. Once the batched data are in device memory, reprocessing is computationally efficient, such that multiple iterations of the analyses can be run for optimization purposes. We have demonstrated a prototypical Higgs analysis implementation using these computational kernels which can be evaluated on a billion MC and data events in less than an hour with optional GPU offloading. Several optimizations remain possible in a future work, among them optimizing the data access via efficient decompression and caching, scaling across multiple machines as well as optimizing the threading performance. We hope the approach shown here will spark discussion and further development of fast analysis tools which would be useful for scientists involved in HEP data analysis and more widely in data intensive fields.

Acknowledgment

We would like to thank Jim Pivarski and Lindsey Gray for helpful feedback at the start of this project. We are grateful to Nan Lu and Irene Dutta for providing a reference implementation of the $H(\mu\mu)$ analysis that could be adapted to vectorized code. We would like to thank Christina Reissel for being an independent early tester of these approaches and for helpful feedback on this report. The availability of the excellent Python libraries `uproot`, `awkward`, `coffea`, `Numba`, `cupy` and `numpy` was imperative for this project and we are grateful to the developers of those projects. Part of this work was conducted at "*iBanks*", the AI GPU cluster at Caltech. We acknowledge NVIDIA, SuperMicro and the Kavli Foundation for their support of "*iBanks*".

References

[1] I. Antcheva *et al.*, Comput. Phys. Commun. **180**, 2499 (2009) doi:10.1016/j.cpc.2009.08.005 [arXiv:1508.07749 [physics.data-an]].

[2] J. Pata *et al.* DOI: 10.5281/zenodo.3245494

[3] J. Pivarski *et al.* `https://github.com/scikit-hep/uproot`

[4] J. Pivarski *et al.* `https://github.com/scikit-hep/awkward-array`

[5] Lam, S.K., Pitrou, A. and Seibert, S., 2015, November. Numba: A llvm-based python jit compiler. In Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (p. 7). ACM.

[6] Bockelman, B., Zhang, Z. and Pivarski, J., 2018, September. Optimizing ROOT IO For Analysis. In Journal of Physics: Conference Series (Vol. 1085, No. 3, p. 032012). IOP Publishing.